Software-Projekt I

Prof. Dr. Rainer Koschke

Arbeitsgruppe Softwaretechnik
Fachbereich Mathematik und Informatik
Universität Bremen

Sommersemester 2013

Software-Prüfungen I

Software-Prüfungen II

Zusammenfassung der Testarten
Testmanagement
Testplan
Testfallspezifikation
Testschadensbericht
Wiederholungsfragen

Arten der Software-Qualitätssicherung

- **konstruktiv**: entstehende Software bzw. der Entwicklungsprozess besitzen Qualitätseigenschaften a priori
- **analytisch**: diagnostische Maßnahmen, um existierendes Qualitätsniveau zu bestimmen

– Frühauf u. a. (2000)

Arten der Prüfungen

- **Validierung**: Are we doing the right thing?
  - Prüfung, ob das, was entwickelt wird, gewünscht ist.
- **Verifikation**: Are we doing the thing right?
  - Prüfung, ob der Entwicklungsschritt X richtig durchgeführt wurde.

Zur Qualitätssicherung gehören Prüfungen; eine Prüfung in der Entwicklung kann grundsätzlich nach zwei Prinzipien erfolgen.

Vergleich mit einer Fahrt auf der Basis einer Wegbeschreibung:

A: Haben wir das vorgesehene Ziel erreicht?

B: Sind wir, wie es die Beschreibung verlangt, an der richtigen Stelle auf die richtige Straße abgebogen?

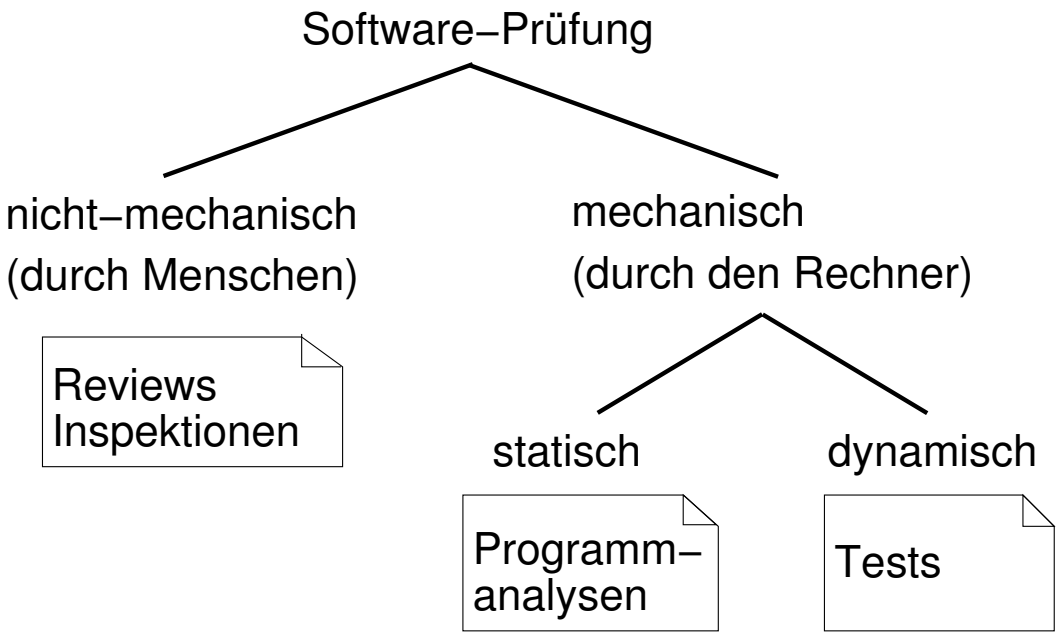
Offenbar sind beide Prüfungen sinnvoll: (A) ergibt die wichtigere Aussage, lässt sich aber nur anwenden, wenn das Endziel oder ein definiertes Zwischenziel erreicht wurde. (B) lässt sich nach jedem Schritt anwenden.

Man beachte aber, dass die Wörter Validierung und Verifikation auch anders gebraucht werden, nämlich

- Validierung als Oberbegriff für alle Prüfungen,
- Verifikation speziell für Prüfungen formaler Art (Programmbeweis und ähnliches).

Zur Vermeidung von Missverständnissen spricht man dann von externer und interner Validierung (entsprechend A und B) bzw. von formaler Verifikation.

Software-Prüfung



Am populärsten ist der Test; aber man kann mit dem Rechner auch statisch prüfen. Fast immer kann man aber inspizieren.

Fragen

- Wozu soll Software getestet werden?
- Was sind die Grenzen des Tests?
- Welche Arten und Varianten des Software-Tests gibt es?
- Wie wählt man eine geeignete Test-Strategie aus?
- Wie erstellt man einen Testplan?
- Wie führt man Tests durch?

N.B.: Diese Darstellung folgt in weiten Teilen Kapitel 11 des Buchs von Brügge und Dutoit (2004).

Ein korrektes Programm?

```
class Kalender {
 public static class MonatUngueltig extends Exception {};
 public static class JahrUngueltig  extends Exception {};
 public static boolean istSchaltJahr(int jahr)
                        {return (jahr % 4) == 0;}
 public static int TageProMonat (int monat, int jahr)
     throws MonatUngueltig, JahrUngueltig {
     int anzTage;
     if (jahr < 1) { throw new JahrUngueltig(); }
     if (monat in {1, 3, 5, 7, 10, 12}) { anzTage = 32;}
     else if (monat in {4, 6, 9, 11})   { anzTage = 30; }
     else if (monat == 2) {
         if (istSchaltJahr (jahr)) anzTage = 29;
         else anzTage = 28;
     } else throw new MonatUngueltig();
     return anzTage;
 }
}
```

Definition
Zuverlässigkeit: Maß für den Erfolg, inwieweit das beobachtete Verhalten mit dem spezifizierten übereinstimmt.

Software-Zuverlässigkeit: Wahrscheinlichkeit, dass ein Software-System während einer festgelegten Zeit unter festgelegten Bedingungen keinen Systemausfall verursachen wird (IEEE Std. 982-1989 1989).

Fehlerbegriff

Definition
Störfall (Ausfall, *Failure*): jegliche Abweichung des beobachteten Verhaltens vom spezifizierten.

Im Nicht-Schaltjahr 200 wird für den Monat Februar 29 ausgegeben.

Fehlerhafter Zustand (*Fault*): Zustand, in dem ein Weiterlaufen des Systems zu einem Störfall führen würde.

Die Funktion `istSchaltjahr(200)` liefert `true`.

Fehler (*Defect*, *Error*): mechanische oder algorithmische Ursache eines fehlerhaften Zustands.

Die Prüfung in `istSchaltjahr` ist falsch.

Mechanische Ursache: Fehler in der virtuellen Maschine (Plattform); z.B. Bug in Java-Virtual-Machine, Compiler oder Hardwaredefekt.

Algorithmische Ursache: Fehler im Algorithmus, z.B. Off-by-One-Fehler, uninitialisierte Variable,

Performanzengpässe aufgrund eines schlechten Entwurfs.

Testen und seine Begriffe

Definition

Test: systematischer Versuch, in der implementierten Software Defekte zu finden.

Erfolgreicher (positiver) Test: Test, der Defekt aufgedeckt hat.

Erfolgloser (negativer) Test: Test, der keinen Defekt aufgedeckt hat.

→ Tests sind Experimente zur Falsifikation der Hypothese "System ist korrekt".

→ Aus negativem Test folgt noch lange nicht, dass kein Defekt vorhanden ist.

Der Test und seine Verwandten

Tests sind nur *ein* Mittel, die Zuverlässigkeit zu steigern:

- Fehlervermeidung (z.B. Entwicklungsmethoden, Verifikation, statische Analyse)
- Fehlerentdeckung (z.B. Tests, `assert`, "Quality-Feedback-Agent", Flugschreiber)
- Fehlertoleranz: Behandlung von Fehlern zur Laufzeit, um Programm fortzusetzen

Soziologie des Testens

- Testen wird häufig (aber zu unrecht) als niedere Arbeit angesehen

→ Frischlinge werden zu Testern

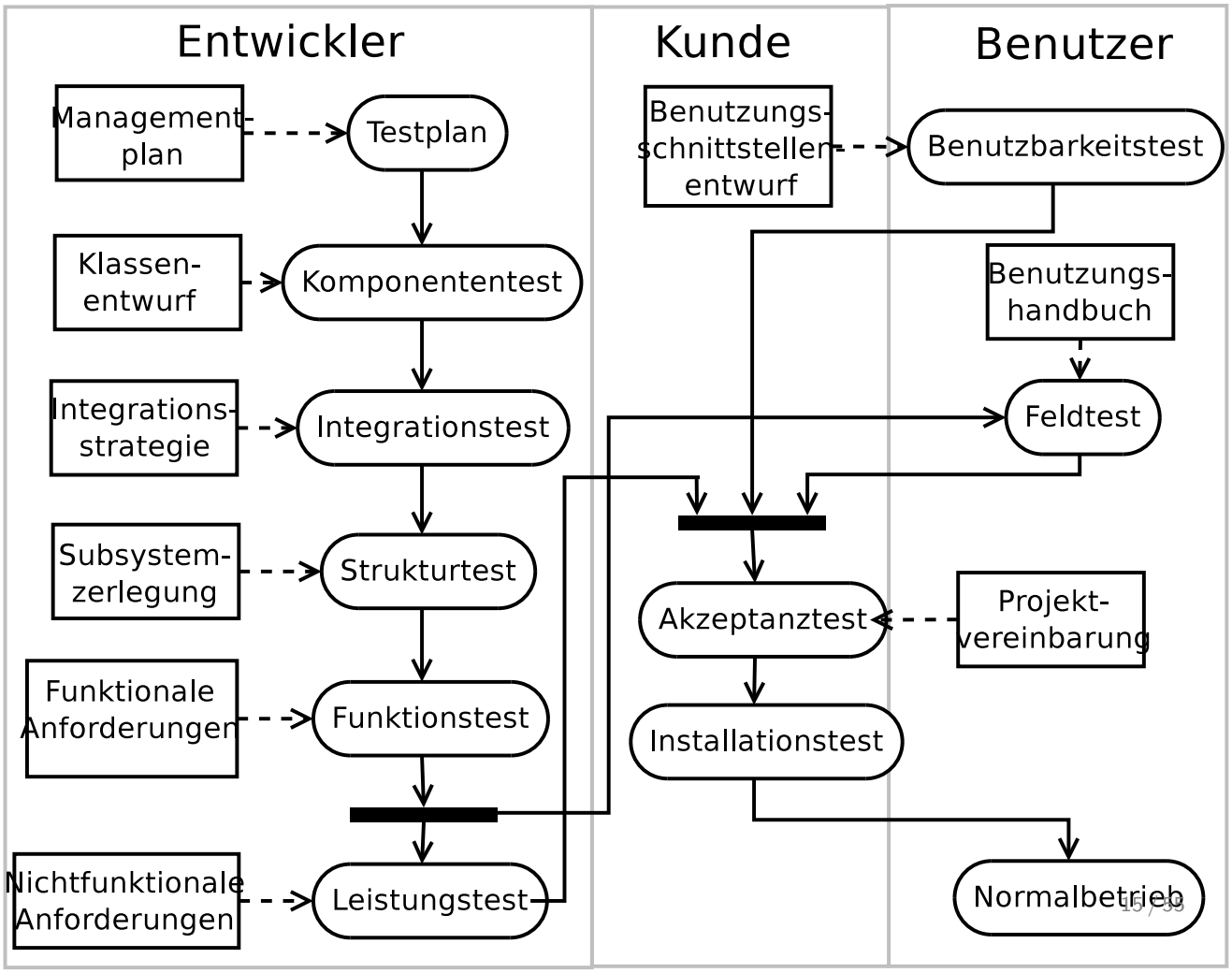
- Tester brauchen jedoch ein umfassendes Systemverständnis (Anforderungen, Entwurf, Implementierung)
- Tester brauchen darüber hinaus Wissen über Prüftechniken

- Autoren haben eine Lesart der Spezifikation verinnerlicht
- Denkfehler in der Implementierung werden sich bei Erstellung von Testfällen wiederholen

→ Tester sollten nicht gleichzeitig Autoren sein

- Tester versuchen, Fehler zu finden
- das Produkt wird kritisiert, dann fühlen sich Autoren selbst kritisiert

→ Egoless-Programming (eine schöne Illusion)

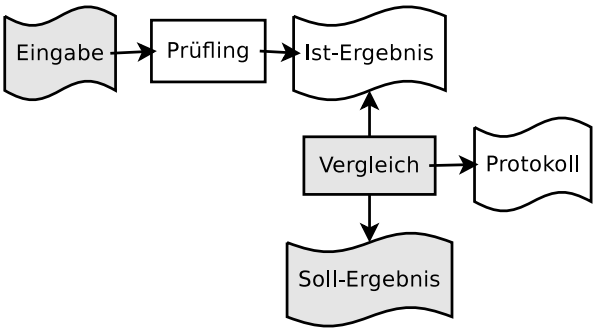


Quelle: Brügge und Dutoit (2004)

- Testplanung: Betriebsmittelzuteilung und Zeitplan
- Benutzbarkeitstest: Fehler im Benutzerschnittstellenentwurf aufdecken; siehe (Papier-)Prototypen
- Komponententest: Fehler in einzelner Komponente (Klasse, Paket o.Ä.) aufdecken
- Integrationstest: Fehler im Zusammenspiel von Komponenten aufdecken
- Strukturtest: Integrationstest mit allen Komponenten (abschließender und vollständiger Integrationstest)
- Systemtest: Test aller in einem System zusammengefasster Subsysteme mit dem Ziel, Fehler bezüglich der Szenarien aus der problemstellung sowie der Anforderungen und Entwurfsziele, die in der Analyse bzw. im Systementwurf identifiziert wurden, zu finden
- Funktionstest: testet die Anforderungen aus dem Lastenheft und die Beschreibung des Benutzerhandbuchs
- Leistungstest: Test der Leistungsanforderungen (Performanz und Speicherbedarf)
- Installationstest: Test der Installation beim Kunden (evtl. mit der Hilfe der Entwickler)
- Akzeptanztest: Prüfung bzgl. Projektvereinbarung durch den Kunden (evtl. mit der Hilfe der Entwickler)

Testfall

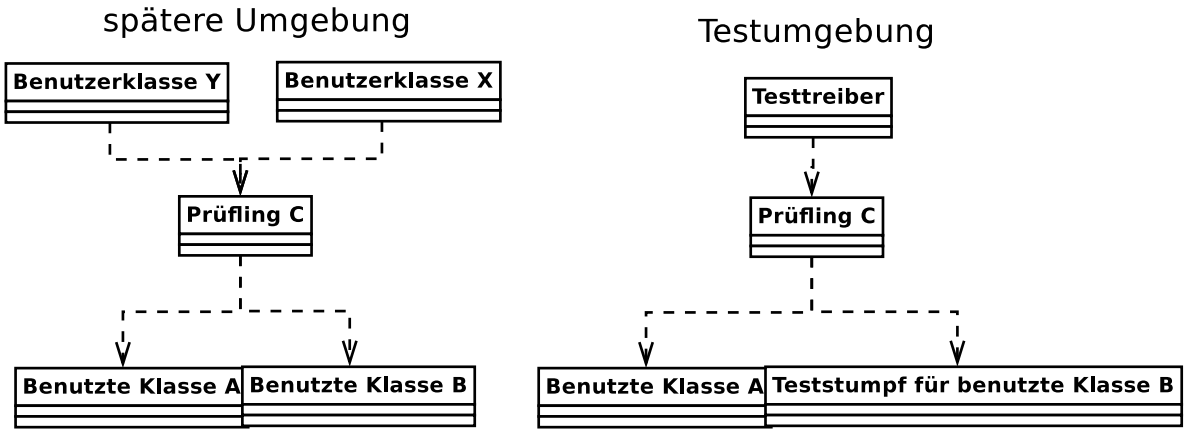
- Name
- Ort
- Eingabe
- Orakel
- Protokoll



Ein Testfall besteht aus:

- Name
- Ort: vollständiger Pfadname des lauffähigen Testprogramms
- Eingabe: Eingabedaten oder Befehle
- Orakel: erwartete Testergebnisse, die mit den aktuellen Testergebnissen des Testlaufs verglichen werden
- Protokoll: gesamte Ausgabe, die durch Test erzeugt wird

Komponententest: Teststümpfe und -treiber



Bestreben: Komponenten so früh wie möglich testen.

Problem: die benutzenden Komponenten und die benutzten Komponenten existieren möglicherweise noch nicht.

Lösung: Testtreiber und -stümpfe

- Prüfling/Testkomponente: die zu testende Klasse/Komponente
- Testtreiber: simuliert den Teil des Systems, der die Testkomponente benutzt
- Teststumpf: simuliert die Komponenten, die die Testkomponente benutzt

In vielen Fällen: Aufwand für Teststumpf entspricht Aufwand für die eigentliche Komponente → Bottom-Up-Entwicklung der Komponenten

Typen von Komponententests

Definition
Black-Box-Tests: Komponente wird nur mit Kenntnis der Schnittstellenbeschreibung entworfen (Implementierung unbekannt).

Techniken:

- Äquivalenztest
- Grenztest
- (zustandsbasierter Test)
- ...

Typen von Komponententests

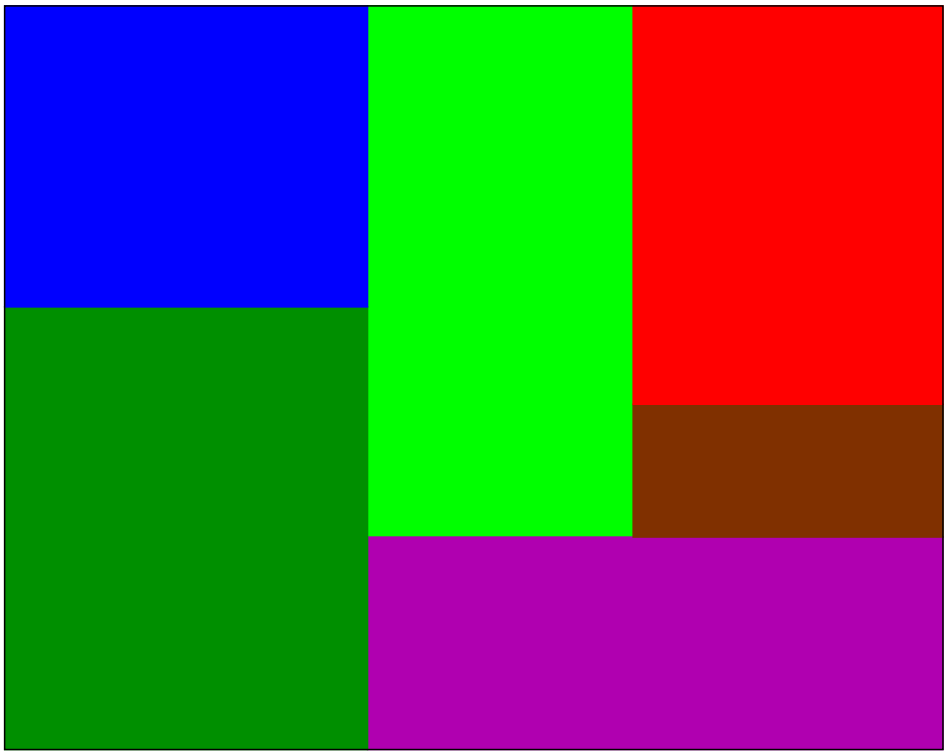
Definition
White-/Glass-Box-Tests: Testfall wird mit Kenntnis der Implementierung entworfen.

Techniken:

- Pfadtest
- (zustandsbasierter Test)
- ...

Äquivalenztest

Ziel: Testfälle minimieren.



Idee:

- Äquivalente Eingaben werden zusammengefasst.
- Zwei Eingaben sind äquivalent, wenn sich der Prüfling *äquivalent* verhält.
- Ein Testfall wird als Repräsentant der Äquivalenzklasse aufgestellt.

Kriterien:

- Abdeckung: Jede mögliche Eingabe gehört zu einer der Äquivalenzklassen
- Disjunktion: Keine Eingabe gehört zu mehr als einer einzigen Äquivalenzklasse
- Repräsentation (die Hoffnung): Falls Ausführung einen fehlerhaften Zustand anzeigt, sobald ein bestimmtes Mitglied einer Äquivalenzklasse als Eingabe benutzt wird, dann kann derselbe fehlerhafte Zustand entdeckt werden, wenn irgendein anderes Mitglied dieser Äquivalenzklasse als Eingabe verwendet wird

Ein korrektes Programm?

```
class Kalender {
 public static class MonatUngueltig extends Exception {};
 public static class JahrUngueltig  extends Exception {};
 public static boolean istSchaltJahr(int jahr)
                        {return (jahr % 4) == 0;}
 public static int TageProMonat (int monat, int jahr)
     throws MonatUngueltig, JahrUngueltig {
     int anzTage;
     if (jahr < 1) { throw new JahrUngueltig(); }
     if (monat in {1, 3, 5, 7, 10, 12}) { anzTage = 32;}
     else if (monat in {4, 6, 9, 11})   { anzTage = 30; }
     else if (monat == 2) {
         if (istSchaltJahr (jahr)) anzTage = 29;
         else anzTage = 28;
     } else throw new MonatUngueltig();
     return anzTage;
 }
}
```


Beispielspezifikation

`Kalender.TageProMonat(monat,jahr)` liefere die Tage eines Monats wie folgt:

- `monat` $\in \{1, 3, 5, 7, 8, 10, 12\} \Rightarrow 31$
- `monat` $\in \{4, 6, 9, 11\} \Rightarrow 30$
- `jahr` ist ein Schaltjahr $\wedge$ `monat` $= 2 \Rightarrow 29$
- `jahr` ist kein Schaltjahr $\wedge$ `monat` $= 2 \Rightarrow 28$

Fehlerfall

- `monat` $< 1 \vee$ *monat* $> 12 \Rightarrow$ `throw MonatUngueltig`
- `jahr` $< 0 \Rightarrow$ `throw JahrUngueltig`

Beispieltestfälle

Äquivalenzklasse	`monat`	`jahr`	Soll
31-Tage-Monat, Nicht-/Schaltjahr	7	1904	31
30 Tage-Monat, Nicht-/Schaltjahr	6	1904	30
Februar, Nicht-Schaltjahr	2	1901	28
Februar, Schaltjahr	2	1904	29
`monat` inkorrekt, `jahr` korrekt	17	1901	`MonatUngueltig`
`monat` korrekt, `jahr` inkorrekt	7	-1901	`JahrUngueltig`

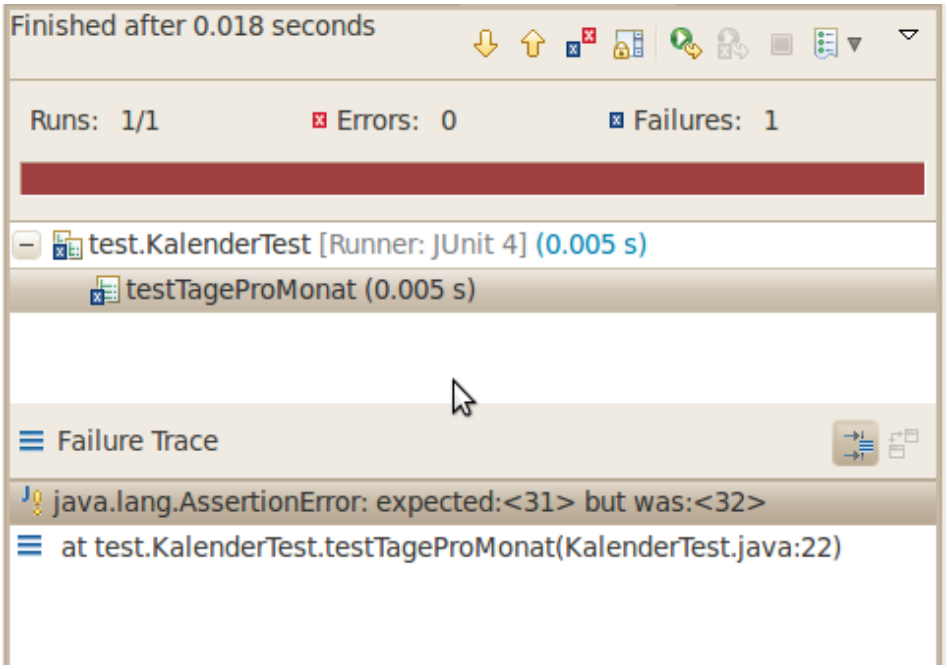
Spezifikationslücke:
`monat` inkorrekt, `jahr` inkorrekt $\Rightarrow$ `MonatUngueltig`
$\vee$`JahrUngueltig`?

Komponententest mit JUnit

```
import static org.junit.Assert.*;
import org.junit.Test;

public class KalenderTest {
  @Test
  public void testTageProMonat1() {
    try {
      assertEquals(31, Kalender.TageProMonat(7, 1901));
    } catch (MonatUngueltig e) {
      fail("Gültiger Monat ergibt Ausnahme: Monat ungueltig.");
    } catch (JahrUngueltig e) {
      fail("Gültiges Jahr ergibt Ausnahme: Jahr ungueltig.");
    }
  }
}
```

Komponententest mit JUnit (in Eclipse)

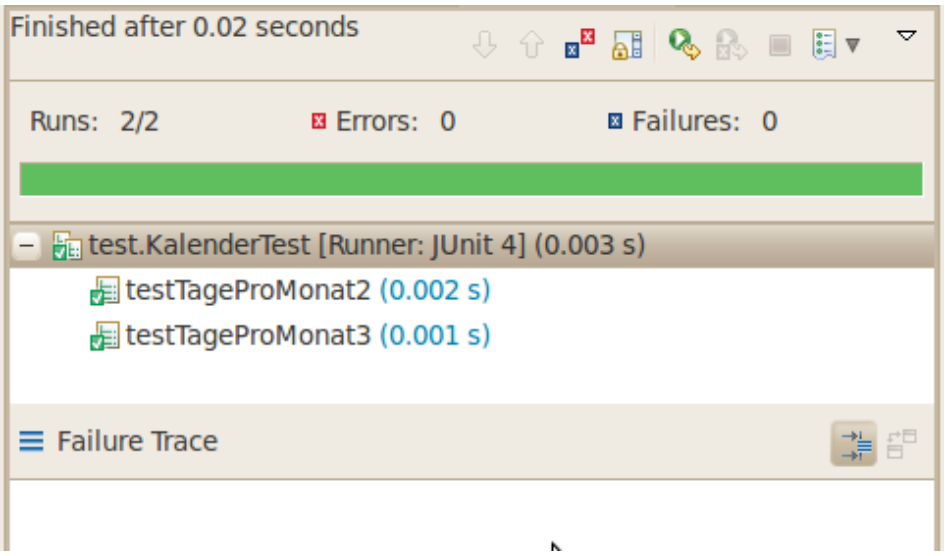


Komponententest mit JUnit

```
@Test(expected=MonatUngueltig.class)
public void testTageProMonat2() throws Exception {
  Kalender.TageProMonat(13, 1901);
}

@Test(expected=JahrUngueltig.class)
public void testTageProMonat3() throws Exception {
  Kalender.TageProMonat(12, -1901);
}
```

Komponententest mit JUnit (in Eclipse)



Grenztest

Beobachtung: “Off-by-one”-Fehler kommen häufig vor.

Idee: Grenzen (Randbereiche, Sonderfälle) von Äquivalenzklassen testen.

Schaltjahrregel: Ein Jahr ist ein Schaltjahr, wenn

- es durch 4 teilbar ist,
- es sei denn, es ist durch 100 teilbar,
- es sei denn, es ist durch 400 teilbar

2000 ist ein Schaltjahr, 1900 ist kein Schaltjahr.

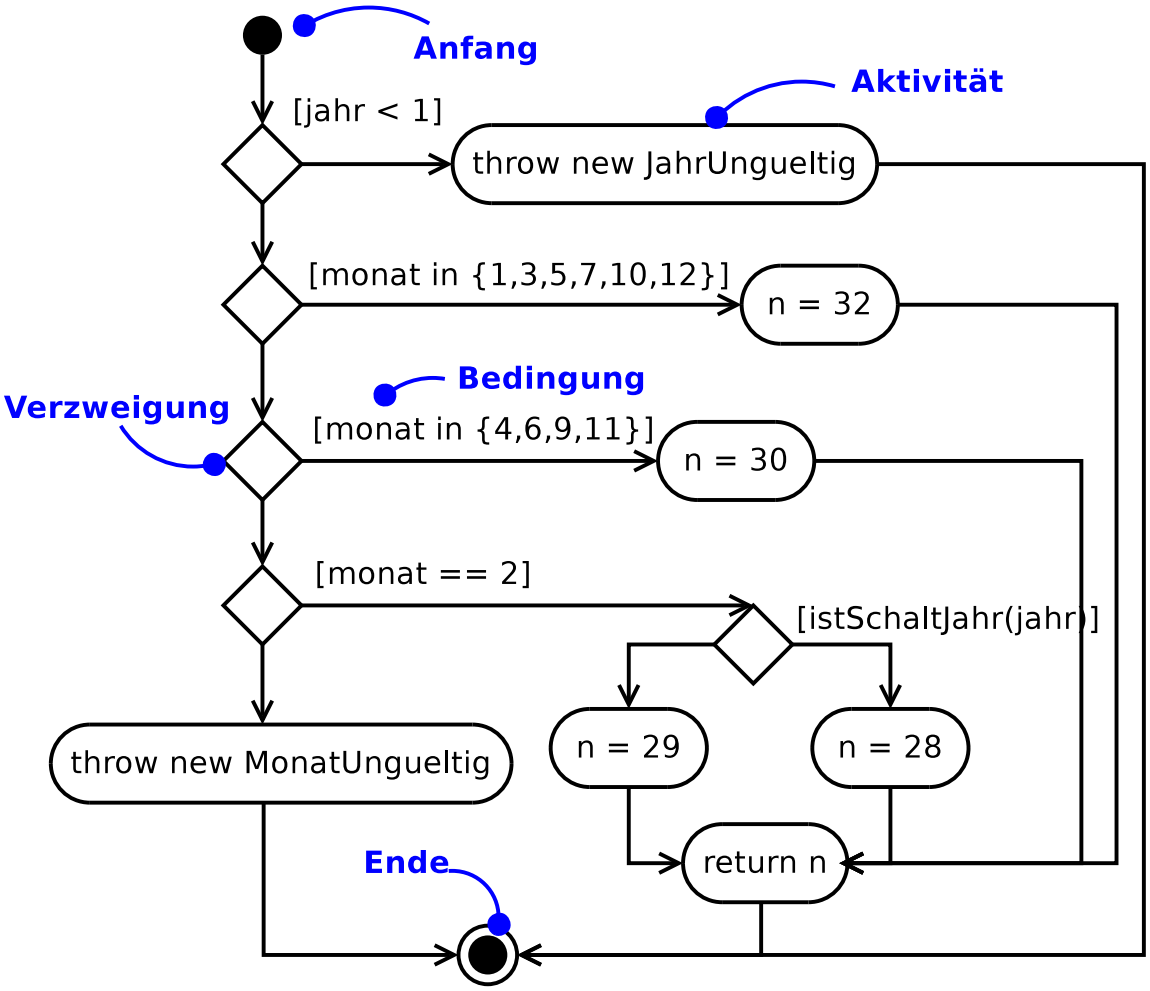
Grenztest

Äquivalenzklasse	`monat`	`jahr`	Soll
Schaltjahre teilbar durch 400	2	2000	29
Nicht-Schaltjahre teilbar durch 100	2	1900	28
gültige Monate	1	2000	31
gültige Monate	12	2000	31
Nichtpositive ungültige Monate	0	1234	`MonatUngueltig`
Positive ungültige Monate	13	1234	`MonatUngueltig`
gültiges Jahr	2	0	29
gültiges Jahr	3	Max-Int	31
ungültiges Jahr	2	-1	`JahrUngueltig`

Pfadtest

Ziel: Testfälle sollten alle Code-Teile testen.

Idee: Konstruiere Testfälle, die jeden möglichen Pfad mindestens einmal ausführen.



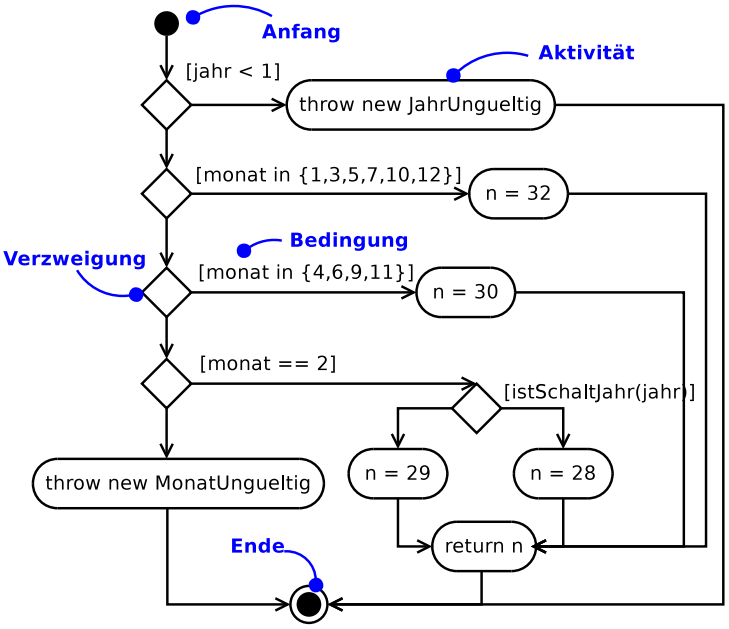
Ausgangspunkt: Kontrollflussgraph (KFG) pro Methode.

- Knoten: Anweisungen und Prädikate
- zwei weitere Knoten: eindeutiger Anfang und Ende einer Methode
- Kanten: Kontrollfluss zwischen Anweisungen (unbedingt) und Prädikaten (bedingt)
- Pfad: Menge von Kanten, die vom Anfang zum Ende des KFGs führen

Maße der Testabdeckung

C0, Anweisungsüberdeckung

(Befehlsabdeckung/Statement-Coverage): Verhältnis von Anzahl der mit Testdaten durchlaufenen Anweisungen zur Gesamtanzahl der Anweisungen.

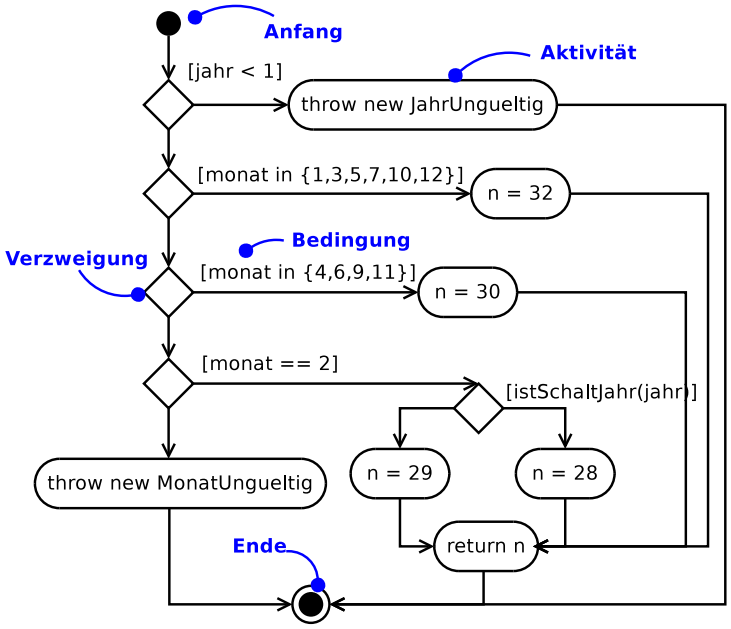


Befehlsabdeckung / statement coverage / C0-Abdeckung: Gesucht ist eine Menge von Testfällen, die jeden Befehl in dem zu testenden Code mindestens einmal ausführen. Dieses Kriterium ist sehr schwach, denn es werden in der Regel nur sehr wenige Fehler (20%) gefunden. Fehler entstehen meist dadurch, dass Befehle in bestimmten Zuständen ausgeführt werden und nicht nur in irgendeinem.

Ein Code-Coverage-Tool kann helfen, um zu sehen, welche Befehle nicht oder selten ausgeführt werden. Denn ein notwendiges Kriterium ist die Befehlsabdeckung allemal.

Maße der Testabdeckung

C1, Zweig-/Entscheidungsüberdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Zweige zur Gesamtanzahl der Zweige.



Jedes Prädikat muss im Code mindestens einmal wahr und einmal falsch sein. Schließt die Befehlabdeckung mit ein (es sei denn, es gibt unerreichbare Befehle) und ist äquivalent zur Befehlsabdeckung, wenn jeder wahr/falsch-Fall auch mindestens einen Befehl beinhaltet. Dieses Kriterium ist weiterhin relativ schwach. Erfahrungsgemäß 25% Fehlerfindung.

Maße der Testabdeckung

C2, Bedingungsabdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Terme innerhalb von Entscheidungen zur Gesamtanzahl der Terme.

```
i := 1;
loop
   found := a(i) = key;
   exit when found or i = Max_Entries;
   i := i + 1;
end loop;
```


Nur verschieden von C1, wenn logische Operatoren keine Short-Circuit-Operatoren sind (d.h. es werden grundsätzlich alle Operanden eines logischen Operators ausgewertet).

In jedem Prädikat wird jeder Bedingungsteil (Konjunktions- oder Alternationsglied) mindestens einmal wahr und einmal falsch gesetzt.

N.B.: Das umfasst nicht die Entscheidungsfallabdeckung:

`true` **and** `false` → false-Kante
`false` **and** `true` → false-Kante

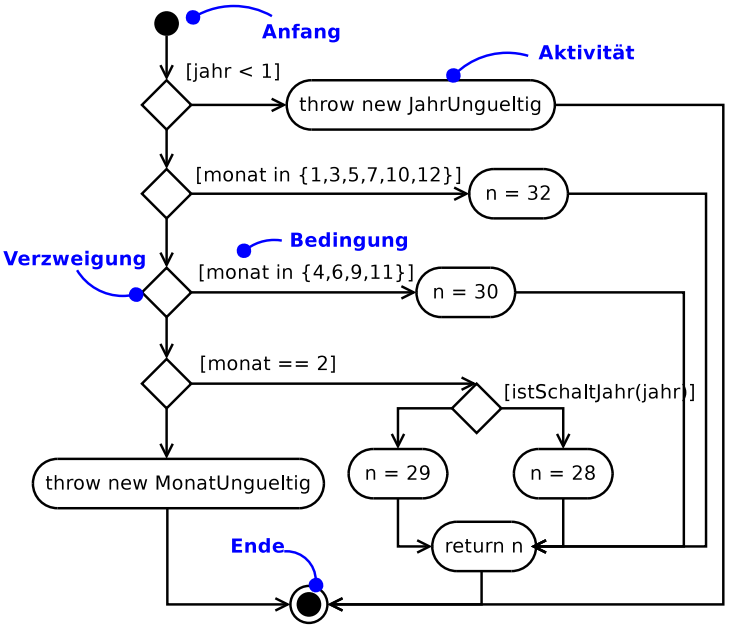
(sofern der Operator **and** kein Short-Circuit-Operator ist).

Man sollte beide kombinieren.

Man bedenke, dass `false` **and** `x` → `false` ist wenn `x` keine Seiteneffekte hat.

Maße der Testabdeckung

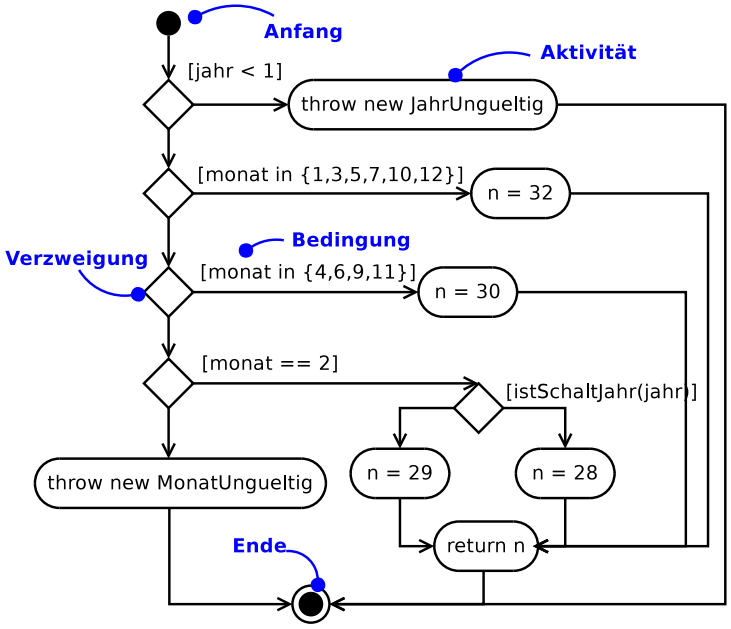
C3, Abdeckung aller Bedingungskombinationen Verhältnis von Anzahl der mit Testdaten durchlaufenen Bedingungskombinationen zur Gesamtanzahl der Bedingungskombinationen.



Bedingungenkombinationsabdeckung / multiple condition coverage: Hier werden Prädikate gemäß einer Wahrheitstabelle alle Bedingungskombinationen aller in allen Kombinationen mal wahr und mal falsch gesetzt. Interessanterweise bringt dies nicht viel mehr an Erfolg, aber viel mehr an Aufwand. ·

Maße der Testabdeckung

C4, Pfadabdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Pfade zur Gesamtanzahl der Pfade.



(Eingeschränkte) Pfadabdeckung: Hier werden auch bei Schleifen mehrere Durchläufe gemacht (keinmal, einmal, zweimal, typische Anzahl, maximale Anzahl). In aller Regel wird nicht auf diese Weise getestet, da der Aufwand sehr groß ist.

Probleme beim Pfadtest

Unrealisierbare Pfade:

```
for (i = 0; i < o.foo(); i++)  // immer o.foo () == 0 ?
{
  x = ...
}

...x... // hat x definierten Wert?

if (p) {...}

if (q) { // p => not q??
  ...
}
```


Polymorphismustest

```
class T {public int foo();}
class NT extends T {public int foo();}
class Factory { T create(int i);}

class UnderTest {
  void bar (int i)
  { T t = (new Factory).create(i);

    if (t.foo() > 0)
       doSomething();
  }
}
```

Methode `bar` kann nicht in Isolation getestet werden. Welches `foo` im Beispiel ausgeführt wird, hängt vom dynamischen Typ von `t` ab. Davon hängen wiederum die weiteren Pfade ab.

Zustandsbasiertes Testen

Verhalten von Komponente hängt ab von

- der Eingabe
- ihrem Zustand

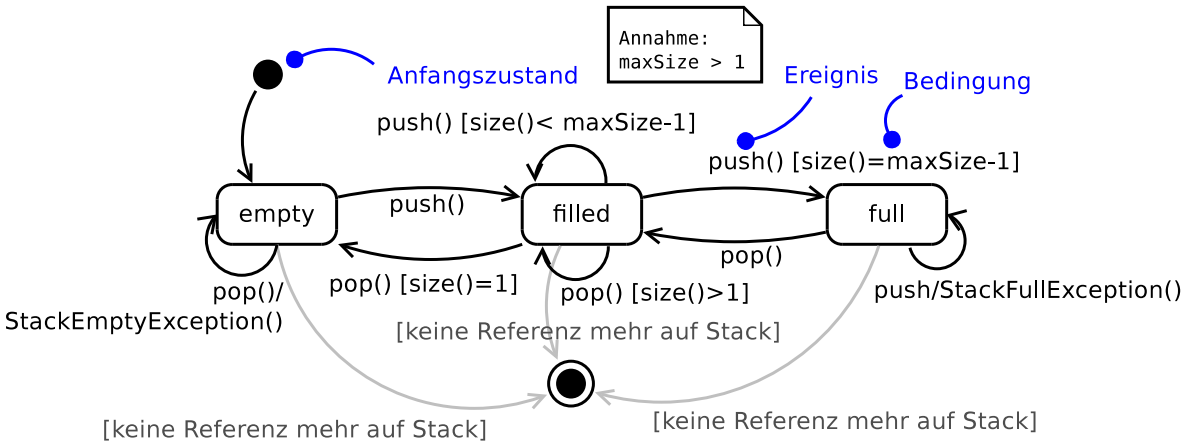
```
class Stack {
    public static class EmptyStack extends Exception {};
    public static class FullStack  extends Exception {};

    public Stack(int maxSize) {...};

    public boolean isEmpty() {...};
    public int size() {...};

    public Object top () throws EmptyStack {...};
    public void push (Object o)  throws FullStack {...};
    public void pop ()   throws EmptyStack {...};
}
```

Zustände eines Stacks



Für alle Zustände einer Komponente:

- Komponente wird in zu testenden Zustand gebracht
- Test für alle möglichen Stimuli:
  - korrekte Eingaben,
  - fehlerhafte Eingaben
  - und Aktionen, die Zustandsübergänge bewirken

Zustandsbasiertes Testen I

```
public class StackStateTest {

  public final int max = 100;

  public boolean isEmpty(Stack s) {
    return s.size() == 0;
  }

  public boolean isFilled(Stack s) {
    return s.size() > 0 && s.size() < max;
  }

  public boolean isFull(Stack s) {
    return s.size() == max;
  }
}
```


Zustandsbasiertes Testen II

```
public class StackFullTest extends StackStateTest {
private Stack stack;
private Object last;

@Before
public void setup() throws Exception {
  stack = new Stack(max);
  // put stack into state full
  for (int i=0; i < max; i++) {
    last = new Object();
    stack.push(last);
  }
}

@Test
public void testSize() {
  // test legal action 'size'; no transition
  assertEquals(max, stack.size());
  assertEquals(true, isFull(stack)); }}
```

Zustandsbasiertes Testen III

```
@Test
public void testPush() {
  // test illegal action 'push'
  try {
    stack.push(new Object());
    fail();
  } catch (StackFull e) {
    assertEquals(true, isFull(stack)); }}

@Test
public void testPop() throws StackEmpty {
  // test legal action 'pop'
  assertEquals(last, stack.pop());
  assertEquals(max-1, stack.size());
  assertEquals(true, isFilled(stack));  }}
```


Vergleich von White- und Black-Box-Tests

Eigenschaft	Black-Box-Test	White-Box-Test
Test auf Basis von	Schnittstellen-spezifikation	Lösung
Wiederverwendung bei Änderung der Struktur	ja	eingeschränkt
Geeignet für Testart	alle	Komponententest
Finden Fehler aufgrund von	Abweichung von Spez.	eher Kodierfehler

Black-Box-Tests (Funktionstests) betrachten den Prüfling als schwarze Box. Sie setzen keine Kenntnisse über die Interna voraus.

White-Box-Tests (Strukturtests, Glass-Box-Tests) betrachten Interna des Prüflings, um Testfälle zu entwickeln.

White-Box Methoden eignen sich lediglich für den Modultest, allenfalls noch für den Integrationstest. Die Idee ist es, die Testfälle anhand der inneren Programmstruktur zu finden.

Achtung: White-Box-Testing testet keine fehlenden Pfade oder Anweisungen. Das ist eine echte Schwäche, denn es wird nur das getestet, woran der Programmierer auch wirklich (wenn auch evtl. fehlerhaft) gedacht hat!

Black-Box-Tests können wiederverwendet werden, wenn sich die Struktur, aber nicht das Verhalten ändert.

Black- versus White-Box-Tests

Nicht entweder-oder, sondern sowohl-als-auch!

Komplementäre Verwendung:

1. Erstelle Funktionstest
2. Messe Abdeckung
3. Wenn Abdeckung nicht ausreichend; ergänze durch Strukturtest; zurück zu 2.

Strategien des Integrationstests

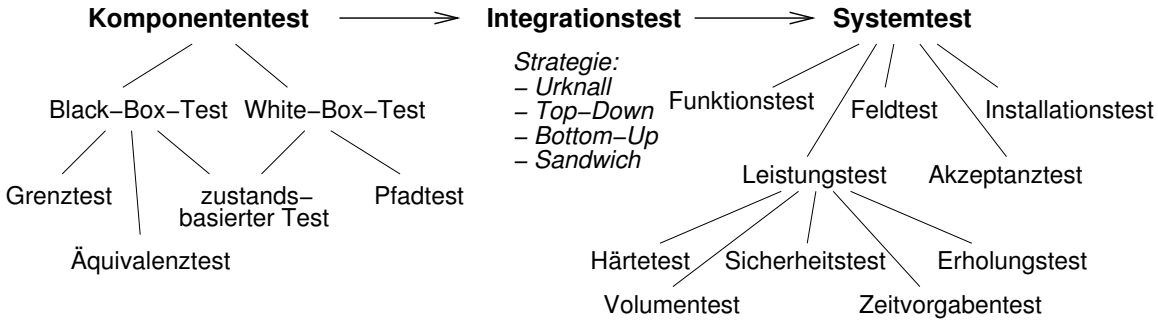
- Urknalltest
- Bottom-Up
- Top-Down
- Sandwich-Test-Strategie

- Urknalltest: alle Komponenten werden einzeln entwickelt und dann in einem Schritt integriert
  → erschwert Fehlersuche
- Bottom-Up: Integration erfolgt inkrementell in umgekehrter Richtung zur Benutzt-Beziehung
  → keine Testrümpfe notwendig (aber Testtreiber)
  → Fehler in der obersten Schicht werden sehr spät entdeckt; die enthalten jedoch die Applikationslogik
- Top-Down: Integration erfolgt in Richtung der Benutzt-Beziehung
  → Fehler in der obersten Schicht werden sehr früh entdeckt
  → Testrümpfe notwendig
- Sandwich-Test-Strategie: Kombination von Top-Down und Bottom-Up
  – Identifikation der zu testenden Schicht: Zielschicht
  – zuerst: individuelle Komponententests
  – dann: kombinierte Schichttests:
    - Oberschichttest mit Zielschicht
    - Zielschicht mit Unterschicht
    - alle Schichten zusammen

Leistungstests

- Härtetest: viele gleichzeitige Anfragen
- Volumentest: große Datenmengen
- Sicherheitstests: Sicherheitslücken aufspüren
- Zeitvorgabentests: werden spezifizierte Antwortzeiten eingehalten?
- Erholungstests: Tests auf Erholung von fehlerhaften Zuständen

Zusammenfassung der Testarten



Testdokumentation: Aufzeichnung der Testaktivitäten

- Testplan: Projektplan fürs Testen
- Testfallspezifikation: Dokumentation eines jeden Testfalls
- Testvorfallbericht (Testprotokoll): Ergebnisse des Tests und Unterschiede zu erwarteten Ergebnissen
- Testübersicht: Auflistung aller Fehler (entdeckt und noch zu untersuchen)
  - → Analyse und Priorisierung aller Fehler und deren Korrekturen

Testplan und Testfallspezifikation können sehr früh schon erstellt werden.

Testplan (IEEE Std. 829-1998 1998)

1. Einführung
2. Beziehung zu anderen Dokumenten
3. Systemüberblick
4. Merkmale, die getestet/nicht getestet werden müssen
5. Abnahmekriterien
6. Vorgehensweise
7. Aufhebung und Wiederaufnahme
8. Zu prüfendes Material (Hardware-/Softwareanforderungen)
9. Testfälle
10. Testzeitplan: Verantwortlichkeiten, Personalausstattung und Weiterbildungsbelange, Risiken und Schadensmöglichkeiten, Zeitplan

(Entspricht einer vereinfachten Variante von IEEE Std. 829-1998 (1998).)

2.: Beziehung zu Anforderungsspezifikation und Architekturbeschreibung; Einführung von Namenskonventionen, um Tests und Anforderungen/Architektur in Beziehung zu setzen.

3.: Überblick über System hinsichtlich jener Komponenten, die durch Komponententests geprüft werden sollen. Granularität der Komponenten und ihre Abhängigkeiten.

4.: Konzentration auf funktionale Gesichtspunkte beim Testen; identifiziert alle merkmale und Kombinationen von Merkmalen, die getestet werden müssen. Beschreibt auch diejenigen Merkmale, die nicht getestet werden und gibt Gründe dafür an.

5.: Spezifiziert Abnahmekriterien für die Tests (z.B. Testabdeckung).

6.: Allgemeine Vorgehensweisen beim Testablauf; Integrationsstrategien.

7.: Kriterien, wann Testaktivitäten ausgesetzt werden. Testaktivitäten, die wiederholt werden müssen, wenn das Testen wieder aufgenommen wird.

8.: Notwendige Betriebsmittel: physikalische Eigenarten der Umgebung sowie Anforderungen an Software, Hardware, Testwerkzeuge und andere Betriebsmittel (z.B. Arbeitsraum).

9.: (das Herzstück des Testplans) Auflistung aller Testfälle anhand individueller Testfallspezifikationen.

Testfallspezifikation

- Testfallbezeichner: eindeutiger Name des Testfalls; am Besten Namenskonventionen benutzen
- Testobjekte: Komponenten (und deren Merkmale), die getestet werden sollen
- Eingabespezifikationen: erwartete Eingaben
- Ausgabespezifikationen: erwartete Ausgaben
- Umgebungserfordernisse: notwendige Software- und Hardware-Plattform sowie Testrümpfe und -stümpfe
- Besondere prozedurale Anforderungen: Einschränkungen wie Zeitvorgaben, Belastung oder Eingreifen durch den Operateur
- Abhängigkeiten zwischen Testfällen

Testschadensbericht

- welche Merkmale wurden getestet?
- wurden sie erfüllt?
- bei Störfällen: wie können sie reproduziert werden?

→ Sammlung und Priorisierung in der Testübersicht
→ Test ≠ Fehlersuche bzw. -korrektur!

Wiederholungsfragen I

- Erläutern Sie die Unterschiede in den Fehlerbegriffen Störfall, fehlerhafter Zustand und Fehler.
- Was ist ein positiver, was ein negativer Testfall?
- Wer sollte testen?
- Welche Arten von Tests gibt es?
- Was ist ein Testfall?
- Welche Aufgabe erfüllen Teststümpfe und -treiber für den Komponententest?
- Was ist der Unterschied von Black- und Glass-Box-Tests?
- Welche Techniken des Komponententests gibt es?
- Erläutern Sie die Idee von Äquivalenztests am konkreten Beispiel.
- Erläutern Sie die Idee von Grenztests am konkreten Beispiel.
- Erläutern Sie die Idee von Pfadtests am konkreten Beispiel.

Wiederholungsfragen II

- Wie benutzt man JUnit für den Komponententest?
- Wozu benötigt man Polymorphismustests?
- Erläutern Sie die Idee von zustandsbasierten Tests am konkreten Beispiel.
- Welche Testabdeckungsmaße gibt es?
- Nennen Sie Strategien für Integrationstests und diskutieren Sie sie.
- Erläutern Sie Leistungstests.
- Was ist ein Testplan?

1 **Brügge und Dutoit 2004** Brügge, Bernd ; Dutoit, Allen H.: Objektorientierte Softwaretechnik. Prentice Hall, 2004

2 **Frühauf u. a. 2000** Frühauf ; Ludewig ; Sandmayr: Software-Prüfung. 4. Auflage. vdf Zürich, 2000

3 **IEEE Std. 829-1998 1998** : IEEE Standard for Software Test Documentation. IEEE Standards Board, IEEE Std. 829-1998. 1998

4 **IEEE Std. 982-1989 1989** : IEEE Standard Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE Standards Board, IEEE Std. 982-1989. Juli 1989